Git Notes

(Be very systematic in following instructions since any deviation, the future steps/explanations will not align with your current state of git. Worst case if you mess up, create a new folder and restart from the beginning!)

Creating local repo

1. “git config” command is used to set or get configuration variables. --global: This flag specifies that you want to set the configuration globally, meaning it will apply to all repositories on your system. user.name: A configuration variable name that identifies you as the author of commits. Similarly you can also set email, to associate the specified email with your commits, so others can contact you.
    a. git config --global user.email "kc@gmail.com"
    b. git config --global user.name "Kameswari Chebrolu"
2. “git init” is a command used to initialize a new Git repository in a “directory”. When you run git init in a directory, Git creates a new repository, and it begins tracking changes to files within that directory. So for this to work, create a directory, enter that directory and then do within the directory git init.
    a. mkdir git-demo
    b. cd git-demo
    c. git init
3. After init, Git creates a new subdirectory named .git in the current directory. This directory contains all the necessary files for the Git repository, including configuration files, object databases etc, You can check this folder and explore it via below
    a. ls -a
    b. ls -a .git/

Adding Files and checking status

1. Inside the git-demo directory, we will now create some files and then add one of it to the staging area and check status. First two commands are creating two files. The third command is only adding file1.txt to the staging area (note we have done no commits yet). The fourth command is showing the status of git so far i.e. what is in staging area (file1.txt) and what files are not being tracked (file2.txt)
    a. echo "v1" > file1.txt
    b. echo "v1" > file2.txt
    c. git add file1.txt
    d. git status

Committing files, checking status and file content

1. Let us now do our first commit. -m option avoids the editor. The commit output may show something like [master f90d290]: indicates the branch you're currently on (master in this case) and the unique identifier of the commit f90d290 which is a SHA-1 hash that

uniquely identifies the commit. Also what files changed, how many inserted, removed etc. Ignore create mode.
    a. git commit -m "first commit: file1.txt whose content is v1"
2. Post this you can do a git status to see how the status changed. You will notice that nothing is under changes to be committed (since whatever is in staging is not committed) but file2.txt is still listed under untracked files
    a. git status
3. Now do git log. It will show details of the first commit (message, author, date, commit id). Note HEAD is a reference to the currently checked-out branch latest commit.
    a. git log
4. Let us make some changes in the folder. Essentially the content of file1.txt changed and we have added another file file3.txt. And then we will move file2.txt to the staging area. Note file1.txt in commit has content v1 and in the working directory v2, rest have content v1. We will then do another commit.
    a. echo "v2" > file1.txt
    b. echo "v1" > file3.txt
    c. git add file2.txt
    d. git commit -m "second commit: file2.txt v1 is committed, file3 not committed. File1 v1 committed, but inside directory we have v2"
5. Let us check what is happening. git status indicates that file1.txt is modified and that change is not committed and file3.txt is untracked. git log shows the 2 commit details. The last three commands show what is inside our latest commit in the currently checked out branch which is master. As you can see, file1.txt in the commit version has “v1” and not “v2”. This highlights the importance of staging i.e. it gives a finer grained control over your commit at file level. If you want to commit file2, but not ready yet to commit file1 changes, all you need to do is add file2 to staging and then proceed to commit. Changes in file1 will not reflect in the committed version.
    a. git status
    b. git log
    c. git show HEAD:file1.txt
    d. git show HEAD:file2.txt
    e. git show HEAD:file3.txt
6. But sometimes you want to commit the entire folder, without moving every file to the staging area. Let us do the following. Let us change file2 to v2, file1 already is at v2 in the folder and then commit using the “-am” flag. With the -a option, Git automatically stages all **tracked files** with changes, so you don't need to use git add. You will notice via the last three commands that file1.txt and file2.txt are v2 now. Note file3 is not committed yet. This is because it is not being tracked. file1/2 were added at some point in the past to staging and are being tracked, hence they were auto-committed via -a. If you want file3 committed, you have to first add it to staging. After this it will be tracked and you can use -a to auto-commit without adding again and again to staging.
    a. echo "v2" > file2.txt
    b. git commit -am "third commit: file1 v2 is commited, so is file2 with v2, file3 still not committed"

c. git status
d. git log
e. git show HEAD:file1.txt
f. git show HEAD:file2.txt
g. git show HEAD:file3.txt

Example that covers what is in the table in slides (26/27)

This is self explanatory for most parts. “cat file1” shows what is in the folder. “git show :file1” shows the content of file1 in the staging area and “git show HEAD:file1” shows content of file1 in the commit (the three columns in the slides)

1. Initialization
    a. mkdir git-another-demo
    b. cd git-another-demo
    c. git init
2. Slide 24
    a. echo "v1" > file1
    b. git add file1
        i. cat file1
        ii. git show :file1
    c. git commit -m "file1 v1 committed"
        i. cat file1
        ii. git show :file1
        iii. git show HEAD:file1
    d. echo "v2" > file1
        i. cat file1
        ii. git show :file1
        iii. git show HEAD:file1
3. Slide 25
    a. git add file1
        i. cat file1
        ii. git show :file1
        iii. git show HEAD:file1
    b. echo "v3" > file1
        i. cat file1
        ii. git show :file1
        iii. git show HEAD:file1
    c. git commit -m "file1 in staging at v2 committed"
        i. cat file1
        ii. git show :file1
        iii. git show HEAD:file1
    d. git commit -am "file1 in working moved to staging as well as committed , all v3"

i. Above command can also be replaced by git commit file1 -m "msg", where you are explicitly specifying file1

Examining difference between files

Lets go back to the git-demo folder again. Remember file1/2 are committed and both are at v2; file3 is not committed yet.

1. Let us change file1 to v3
    a. echo "v3" > file1.txt
2. Let us now compare the diff between the working folder and the latest commit of file1. The first two commands show the content of the file, while the last shows the diff
    a. cat file1.txt
    b. git show HEAD:file1.txt
    c. git diff HEAD
3. Let us now compare the diff between the staged version and the latest commit of file1.
    a. git show :file1
    b. git show HEAD:file1
    c. git diff --cached HEAD

Undoing changes via checkout

Where are we?

1. There are overall 3 commits. Remember file1:v3 in the working folder but what is committed is file1:v2. File2 is in sync, both working and commit are at v2.
    a. git log
    b. cat file1.txt
2. In the below example, commit id has to be replaced by what is in your machine, my first commit id is ending in 4a73d. This command will not work, it will warn that file1:v3 will be overwritten and command aborts
    a. git checkout 4a73d
3. Let us commit first before proceeding with checkout. While we are at it, let me change file2 to v3 and commit this as well.
    a. echo "v3" > file2.txt
    b. git commit -am "fourth commit where file1:v3, file2:v3"
    c. git log
4. Now let us checkout again. Should show file1:v1, and no file2; HEAD is pointing to first commit and is in detached head state! Note master is at the latest commit still. File3 shows still since it is in the folder, but file2 won't since we did not commit it in the first commit.
    a. git checkout 4a73d
    b. cat file1.txt file2.txt
    c. git log
5. Now let us go back to the latest commit. So, we are back at file1 and file2 both at v3. And HEAD has also gone back to pointing master.

    a. git checkout master
    b. ls
    c. cat file1.txt file2.txt
    d. cat git log
6. Suppose I want file1 to belong to some earlier commit (say v1), while file2 stays at v3. So, let us checkout only that file1 from the earlier commit. Then we will commit the desired state and proceed from there in future.
    a. git checkout 4a73d file1
    b. cat file1.txt file2.txt
    c. git commit -am "fifth commit, file1:v1; file2:v3"
    d. git log

Branching and Switching

1. Where are we? Lets go back to the master first, file1:v1, file2: v3. If you try below, it may say you are already on master, if not, it will checkout master
    a. git checkout master
2. Let us see how many branches are there and then create a newbranch. Git branch will show just one branch which is the master. Create a new branch via switch command. Now you are in this new branch. Checkout that all files are as per what is in the master branch. HEAD will be pointing to newbranch
    a. git branch
    b. git switch -c newbranch
    c. cat file1.txt file2.txt
    d. git log
3. Let us make some changes in this new branch now. HEAD is still pointing to newbranch, while master is at the earlier commit.
    a. echo "v4" > file1
    b. git commit -am "file1:v4, file2:v3"
    c. git log
    d. git status
    e. cat file1.txt file2.txt
4. Let us go back to master from this new branch. You will see file1 is at v1 and file2 at v3 as expected. HEAD is pointing to master now.
    a. git checkout master
    b. cat file1.txt file2.txt
    c. git status
    d. git log
5. You can move between two branches, make changes in each and commit. Use checkout to move to master or newbranch as per your requirement.

Merging

Now that we have two branches, let us see how to introduce conflict between the branches and then merge them. Remember in master: file1:v1 and file2:v3. And in newbranch: file1:v4 and file2:v3

1. First let us go to master and then merge the newbranch. This will not result in conflict, it will just update file1 to v4 since master has not seen any new changes post branching. HEAD has gone back to pointing to master which is in sync with newbranch after below commands.
    a. git checkout master
    b. git merge -m "merging" newbranch
    c. cat file1.txt file2.txt
    d. git log
2. Let us introduce conflict. Ensure you are in master. Make some changes in the master branch.
    a. git checkout master
    b. echo "v5" > file1
    c. git commit -am "mainbranch file1:v5 file2:v3
    d. cat file1.txt file2.txt
3. Now let us go to the new branch. And make some changes there as well. Will change file1 to v6 and also change file3 to v2 and commit both
    a. git checkout newbranch
    b. cat file1.txt file2.txt file3.txt
    c. echo "v6" > file1.txt
    d. echo "v2" > file3.txt
    e. git add file3.txt
    f. git commit -am " In newbranch: file1:v6, file2:v3, file3:v2"
4. Let us merge again in master. Git will fail since there is a conflict when merging.
    a. git checkout master
    b. cat file1.txt file2.txt file3.txt
    c. git merge -m "merging" newbranch
5. We now have to decide whether file1 should be v5 or v6. Let us say, we want it to be v6, in which case, let us checkout that version into the master (first command) and then commit everything and then merge. Note master still has no file3 but after merge, file3 will come to master branch. There was no conflict to resolve wrt this file.
    a. git checkout newbranch file1
    b. cat file1.txt file2.txt file3.txt
    c. git commit -am "main file1:v6 file2:v3"
    d. git merge -m "merging" newbranch
    e. ls
    f. cat file1.txt file2.txt file3.txt

Working with remote origin (optional; not part of syllabus)

Assume you have a github account already and assume there is also a repository called CS104-Demo within it (if not create one using the “New” button under repositories). To access the remote github repository from your local machine (assuming linux), you need to do the following. We will use ssh (remote login) to access the repo. Commands below are to be typed in your local machine unless otherwise specified.

1. Generate a ssh key to securely ssh into git to access the repo. “-t” type of key to generate; “ed25519” is the encryption algorithm to use i.e. generate key corresponding to this algorithm. -C is a comment, I just gave me email id as reference.
    a. ssh-keygen -t ed25519 -C "chebrolu@cse.iitb.ac.in"
2. The key will typically be saved in the .ssh folder with extension .pub (public key) and you can access it via
    a. cat ~/.ssh/id_ed25519.pub
3. Now this key has to be entered in github. Go to github settings, go the SSH and GPG keys and add the key there (select New SSH key and essentially cut paste the content of the above .pub file)
4. Then clone the remote repo locally via below command. Clone copies the repository (including commit history) from the remote server; Creates a local folder with the same name as the repository; Checks out the default branch (usually main or master); Sets up a remote named origin, pointing to the original repository; Links the local branch to the remote branch, enabling future pulls and pushes.To know exactly what to type in place of git@github…. Go to the repo, select code, select SSH , you can copy the git@github…. from there
    a. git clone git@github.com:chebrolu/CS104-Demo.git
5. Enter the repo and do some checks and configuration. “git remote -v” confirms your remote origin. Without git config, you cannot commit your local changes to the remote repository. Hence important to do git config
    a. cd CS104-Demo/
    b. ls
    c. git remote -v
    d. git config --global user.email "chebrolu@cse.iitb.ac.in"
    e. git config --global user.name "Kameswari Chebrolu"
6. Suppose some one changed something in the remote repo, you can get the latest changes via below command. Before you start any work locally, you should always use this command.
    a. git pull origin main
    b. git status
    c. ls
7. Normally it may be better for you to work on your own branch and then merge your changes to origin i.e. remote repo. Below, we are creating a new file called features.txt and committing it locally and then pushing these changes to remote repo via push. Post

this, in github one has to approve the changes and merge it, only then can you pull back the changes into the main branch.

a. git checkout -b my-branch
b. echo "extra features" > features.txt
c. git add features.txt
d. git commit -m "features"
e. git push origin my-branch

8. Post push, let us go back to the main branch (note my-branch still exists). If some one approves the earlier commit to remote repo, when you pull, you should see features.txt in the main branch.
    a. git checkout main
    b. git pull origin main
    c. Ls
9. Your typical workflow will be as below:
    a. Switch to my-branch via git checkout my-branch
    b. Pull latest changes via git pull origin main. Note, this updates your local main branch with the latest remote changes.
    c. Move the latest updates from main to your branch via git merge main (this is merging main with your branch; resolve any conflicts if any)
    d. Code away
    e. Then push changes from my-branch to remote repo via git push origin my-branch